MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo No. 954 February 1987

Formalizing Reusable Software Components In the Programmer's Apprentice

by

Charles Rich
Richard C. Waters

Abstract

There has been a long-standing desire in computer science for a way of collecting and using libraries of standard software components. Unfortunately, there has been only limited success in actually doing this. The lack of success stems not from any resistance to the idea, nor from any lack of trying, but rather from the difficulty of choosing an appropriate formalism for representing components.

For a formalism to be maximally useful, it must satisfy five key desiderata: expressiveness, convenient combinability, semantic soundness, machine manipulability, and programming language independence. The Plan Calculus formalism developed as part of the Programmer's Apprentice project satisfies each of these desiderata quite well. It does this by combining the ideas from flowchart schemas, data abstraction, logical formalisms, and program transformations.

The efficacy of the Plan Calculus has been demonstrated in part by a prototype program editor called the Knowledge-based Editor in Emacs. This editor makes it possible for a programmer to construct a program rapidly and reliably by combining components represented as plans.

(To appear in *Software Reuse*, T. Bigerstaff and A. Perlis (eds.).)

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research has been provided in part by the IBM Corporation, in part by the Sperry Corporation, in part by National Science Foundation grant MCS-7912179, and in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-85-K-0124.

The views and conclusions contained in this document are those of the author, and should not be interpreted as representing the policies, either expressed or implied, of the IBM Corporation, of the Sperry Corporation, of the National Science Foundation, or of the Department of Defense.

The Wide Variety of Components

The biggest problem one faces when developing a formalization for components is that there are many different kinds of things which it would be beneficial to express as components. Essentially any kind of knowledge which is shared between distinct programs is a candidate to become a reusable component. As an illustration of the diversity of potential components, consider the following six examples. In each case, it is important to consider not only what information is contained in the component, but also the kinds of information which are not properly part of the component.

Matrix add – the algorithm for adding together two matrices. This algorithm is independent of the data representation for the matrices and the type of matrix elements.

Stack – the stack data structure and its associated operations `PUSH` and `POP`. Both the representation and the operations are independent of the type of stack element.

Filter positive – the idea of selecting the positive elements of a temporal sequence of quantities available in a loop. For example, in the following fragmentary loop, the `if` statement implements a *filter positive*.

```
do ...
  X = ...;
  if X>0 then ... X ...;
end;
```

This idea is independent of the type of sequence element and the sequence creation method. In particular, the idea is applicable to both loops and recursive programs.

Master file system – the idea of having a cluster of programs (reports, updates, audits, etc.) that operate on a single *master* file which is the sole repository for information about some topic. This idea is essentially a set of constraints on the programs and how they interact with the file. It is independent of the kind of data to be stored in the file and the details of the computation to be performed by the programs.

Deadlock free – the idea that a set of asynchronously interacting programs have the property that they are guaranteed not to reach a state where each program is blocked waiting for some other program to act. This idea places restrictions on the ways in which the programs can interact. However, it is independent of the details of computations to be performed by the programs.

Move invariant – the idea that the computation of an expression can be moved from inside of a scope of repetitive execution to outside of the repetitive scope as long as it has no side-effects and all of the values it references are constants within the repetitive scope. This idea is independent of the specific computation being performed by the expression and by the rest of the repetitive scope. In addition, the idea is applicable to both loops and recursive programs.

The example components above differ from each other along many dimensions. *Matrix add* is primarily a computational component which specifies a particular combination of operations, while *stack* is a data component which primarily specifies a particular combination

of data objects. *Matrix add* and *stack* also differ in that *matrix add* is a concrete algorithm while *stack* is much more of an abstract concept. Another dimension of difference between components is that while *matrix add* can be used in a program as a simple subunit, *filter positive* is fragmentary and must be combined together with fragments that generate and use temporal sequences before it can perform a useful computation.

The first three components are all low-level, localized units. In contrast, *master file system* and *deadlock free* are high-level, diffuse concepts which correspond more closely to sets of constraints than to computational units. These two components in turn differ in that *master file system* is a relatively straightforward set of constraints which can be satisfied individually, while *deadlock free* is a property of an entire system of programs which critically depends on each detail of the interaction between the programs.

Move invariant differs from all of the other components in that it is an optimization which is achieved by a standardized transformation *on* programs rather than a standardized computation performed *by* programs.

Desiderata For a Formalization

Many properties are required of a formalization in order for it to be an effective representation for reusable components. The following five desiderata stand out as being of particular importance.

Expressiveness – The formalism must be capable of expressing as many different kinds of components as possible.

Convenient combinability – The methods of combining components must be easy to implement and the properties of combinations should be evident from the properties of the parts.

Semantic soundness – The formalism must be based on a mathematical foundation which allows correctness conditions to be stated for the library of components.

Machine manipulability – It must be possible to manipulate the formalism effectively using computer tools.

Programming language independence – The formalism should not be dependent on the syntax of any particular programming language.

Given the wide range of components which it would be useful to represent, the expressiveness of a formalization is paramount. An important, though hard to assess, aspect of this is *convenience*. It is not sufficient that a formalism merely be *capable* of representing a given component. To be truly useful, the formalization must be able to conveniently represent the component in a straightforward way which supports the other desiderata rather than representing it via a circumlocution which impedes the other desiderata.

Convenient combination properties are also essential, since they are the way in which components are in fact reused. An important part of this is the desire for *fine granularity* in the representation. The goal is to have each component embody only a single idea or design decision so that users have the maximum possible freedom to combine them as they choose.

A firm semantic basis is needed for a good formalization so that it is possible to be certain of what is being represented by a given component and that the combination process

preserves the key properties of components. The semantic basis does not necessarily need to make totally automatic verification possible. Although less convenient, machine-aided (or even manual) verification of library components is sufficient in many situations.

Machine manipulability of a formalization is a key issue. There are thousands of programming ideas which it would be useful to express as components. In order to be able to effectively deal with such a large library, tools need to be developed to support the automatic creation, modification, selection, and combination of components.

A major problem with previous formalisms has been the focus on existing programming languages as the basis for defining reusable components, when there are in fact important differences between the original goals of these programming languages and the goals of work on reusable components. Existing programming languages are designed primarily to express complete programs in a form which is easily readable by the programmer and which can be effectively executed by a machine. In contrast, the challenge in reusability is to express the fragmentary and abstract components out of which complete programs are built.

The most obvious benefit of a language independent formalism is that it makes it possible to represent components in such a way that they can be reused in many different language environments. However, there is an equally important reason for desiring language independence. In general, a language dependent formalism forces components to be represented in terms of specific control flow and data flow constructs. However, these constructs are typically not an essential part of the component and may limit the way in which it can be combined with other components. For example, if you specifying the `PUSH` operation for a stack in a language dependent way, you typically have to specify particular variable names to be used and whether the operation should be coded in-line or out-of-line when it is used.

Approaches To Formalization

The following sections discuss a number of approaches to the formalization of components. The relative strengths and weaknesses of the approaches are evaluated in the light of the five desiderata. The central theme which ties the sections together is the search for formalisms that are capable of expressing the wide range of components desired without sacrificing the other desiderata. Figure 1 summarizes graphically the major flow of ideas between the approaches discussed.

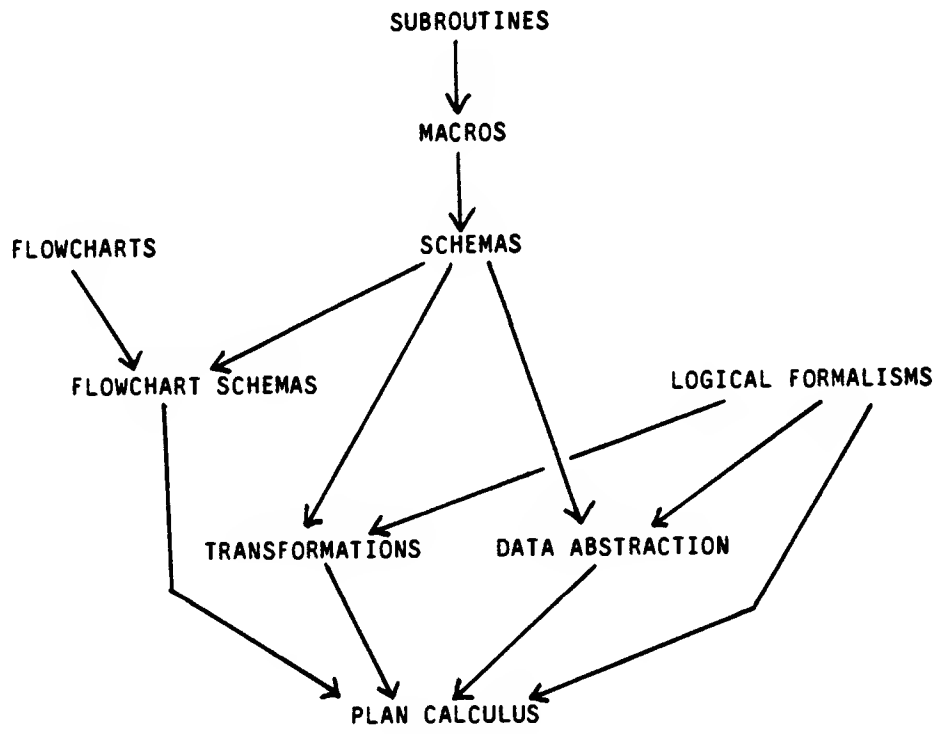

Figure 1: Approaches to Formalization.

As a point of comparison for other formalisms, one must consider free-form English text. Much of the knowledge which needs to be formalized is already captured informally in the vocabulary of programmers and in text books on programming (e.g., [1,14]). The great strength of English text is expressiveness. It is capable of representing any kind of component. Moreover, it is programming language independent. Unfortunately, English text does not satisfy any of the other desiderata. There is no theory of how to combine textual fragments together; there is no semantic basis that makes it possible to determine whether or not a piece of English text means what you think it means; and free-form English text is not machine manipulable in any significant way.

Subroutines

Subroutines have many advantages as a representation for components. They can be easily combined by writing programs which call them. They are machine manipulable in that high-level language compilers and linkage editors directly support their combination. Further, they have a firm semantic basis via the semantics of the programming language

they are written in.

Unfortunately, subroutines are limited in their expressiveness. They are really only convenient for expressing localized computational algorithms such as *matrix add*. They cannot represent data components such as *stack*, fragmentary components such as *filter positive*, diffuse high-level components such as *master file system*, or transformational components such as *move invariant*. In addition, they lack fineness of granularity. It is difficult to write a subroutine without gratuitously specifying numerous details that are not properly part of the component. For example, in most languages, there is no convenient way to write a subroutine representing *matrix add* without specifying the data representation for the matrices and the numbers in them.

Macros

A subroutine specifies a fixed piece of program text corresponding to a component. The only variability allowed is in the arguments which are passed to the subroutine. In contrast, a macro specifies an arbitrary computation which is used to create a piece of program text corresponding to a use of a component. Due to the provision for arbitrary computation, macros are a considerable improvement over subroutines in expressiveness. They can be used to represent data components and fragmentary components. In addition, they can represent components at a much finer granularity. For example, it is straightforward to write a macro which represents *matrix add* independent of the data structures it operates on. Note however, that macros are still not suited to representing diffuse components or transformational ones.

Like subroutines, macros are machine manipulable in that macro processors directly support the evaluation of macro calls and the integration of the resulting program text into the program as a whole. Unfortunately, macros are less satisfactory than subroutines in other respects. Though macro calls are combined syntactically in essentially the same way as subroutine calls, their combination properties are not as simple. For example, since a macro can perform arbitrary computation utilizing its calling form in order to create the resulting program text, there is no guarantee that nested macro calls will operate as they are intended. The macro writer must take extreme care in order to insure that flexible combination is possible. This unfortunately militates against the increased expressiveness which is the primary advantage of macros.

The paramount problem with macros is that they lack any firm semantic basis. Because they allow arbitrary computation, it is very difficult to verify that a macro accurately represents a given component. It is even more difficult to show that a pair of macros can be combined without destructive interaction.

Program Schemas

There has been a considerable amount of theoretical investigation into the benefits of program schemas as a vehicle for representing components[4,9,28]. Program schemas are essentially templates with holes in them which can be filled in with user supplied program text. As such, they can be viewed as a compromise between subroutines and macros. The main improvement of program schemas over macros is that, like subroutines, they have a firm semantic foundation in the semantics of the programming language they are written in and their combination properties are relatively straightforward.

There has not been very much activity directed towards creating an actual programming environment incorporating a library of program schemas. However, there is no reason to believe that program schemas are not at least as machine manipulable as macros. For example, one could create a programming environment supporting program schemas by taking a standard macro processor and limiting the macros that could be written to ones which were essentially program schemas.

Unfortunately, though program schemas are an improvement in expressiveness over subroutines, they are significantly less expressive than macros. Like macros, program schemas are of some use in representing data components such as *stack* and can represent components at a finer granularity than subroutines. In addition, they could be used to represent *matrix add* independent of the representation for the matrices it operates on. However, unlike macros, program schemas cannot in general be used to represent fragmentary components such as *filter positive*. Going beyond this, they are no more useful than macros at representing diffuse or transformational components.

Flowcharts and Flowchart Schemas

A limitation shared by subroutines, macros, and program schemas is programming language dependence. One way to alleviate this problem would be to write components in a programming language independent representation such as a flowchart. Flowcharts use boxes and control flow arrows in order to specify control flow independent of any particular control flow construct. Similarly, data flow arrows can be used to represent data flow independent of any particular data flow construct[8].

A flowchart using data and control flow arrows is basically equivalent to a subroutine and has the same level of expressiveness. In analogy to program schemas one can gain additional expressiveness by using flowchart schemas[13,17]—flowchart templates with holes in them where other flowcharts can be inserted. Just as a programming language can be given a rigorous semantic foundation, a flowchart language can be given a semantic foundation which can serve as a semantic basis for components. In addition, flowcharts and flowchart schemas can be combined together in the same semantically clean way that subroutines and program schemas can be.

To date, flowcharts with data and control flow arrows have primarily been used as a documentation and design aid and have not been given much machine support. However, there is no reason why they cannot be represented in a machine manipulable form and used as part of a programming environment. All that is needed are modules which can translate back and forth between flowcharts and various programming languages.

Flowcharts and flowchart schemas are a significant improvement over subroutines and program schemas in that they are programming language independent. However, with regard to the other desiderata, they are basically identical to subroutines and program schemas. In particular, they are no more expressive. As a result, they are still not fully satisfactory as a representation for reusable components.

Logical Formalisms

With the exception of some macros, the formalisms discussed above are all *algorithmic* in that they represent a component by giving an example (or template) of it in a programming (or flowchart) language. In addition, the only way to use a component is to place it

somewhere in a program. This fundamentally limits the expressiveness of these formalisms. They can only represent localized algorithmic components because the languages being used are only capable of representing algorithms and the way the components are used requires them to be localized.

The extensive work on specifying the semantics of programming languages suggests a completely different approach to the problem of specifying components: using logical formalisms (e.g., the predicate calculus) to represent components. A key advantage of logical formalisms is semantic soundness. (In the role of providing a semantic basis for programming languages, logical formalisms are the ultimate semantic basis for all the formalisms discussed above.) An implicit part of this is that logical specifications must be provided for components so that they can be verified (by hand if necessary).

Another important advantage of logical formalisms is in the area of expressiveness. In contrast to the algorithmic formalisms, logical formalisms have no trouble representing diffuse, high-level components such as *master file system* and *deadlock free*. The usefulness of such components is enhanced by the fact that logical formalisms also have very convenient combination properties. Specifically, the theory generated by the union of two axiom systems is always either the union of the theories of the two component systems or a contradiction, but never some third, unanticipated theory. An additional advantage of logical formalisms is that they are inherently programming language independent.

However, logical formalisms are quite cumbersome when it comes to representing an algorithmic component such as *matrix add* (as opposed to representing the specification for an algorithmic component). Given a component such as *stack*, which combines some non-algorithmic aspects with some algorithmic aspects, logical formalisms are convenient for the former, but not the latter. The above suggests that logical formalisms are best used as an adjunct to, rather than a replacement for, algorithmic formalisms.

The greatest weakness of logical formalisms is in the area of machine manipulability. It is not hard to represent logical formulas in a machine manipulable way. However, at the current state of the art, practical automatic theorem provers are only capable of relatively simple logical deductions. As a result, it is hard to do anything useful with logical formulas. For example, if a programming system were to be based on the combination of components represented as logical formulas, the system would need to have a module which could produce program text corresponding to sets of logical formulas. Unfortunately, although this kind of automatic programming has been demonstrated on small examples [18] it has not yet progressed to the point where it is at all practical.

This problem again suggests that it might be fruitful to combine logical and algorithmic formalisms in order to reduce the amount of deduction which must be performed. Unfortunately, it is not clear how helpful this can be with regard to components such as *master file system* and *deadlock free* which have little or no algorithmic aspects. Assumedly, if one includes *deadlock free* as one of the components describing a set of programs one would like the programming system to be of some assistance in producing programs which are safe from deadlock, or at the very least, be able to detect when deadlock is possible. However, it is not clear that even the latter goal is achievable given the current state of the art of automatic theorem proving.

Data Abstraction

An interesting area of inquiry which has combined logical and algorithmic formalisms is data abstraction. The contribution of data abstractions is that they extend the expressiveness of algorithmic formalizations into the realm of components with data structure aspects. For example, data abstraction can be used to represent *stack* in full generality and to represent *matrix add* without specifying the data representation to be used for the matrices.

A considerable amount of research has been done on how to state the specifications for a data structure and its associated access functions[10,12,16]. This provides a semantic basis for data abstractions and for methods of combining them. In addition, languages such as Alphard[29], CLU[15], and Ada[30] have been developed which have constructs which directly support data abstraction. This demonstrates the ease with which data abstractions can be represented in a machine manipulable (though language dependent) form.

Program Transformations

Neither algorithmic nor logical formalisms are particularly well suited to representing components like *move invariant*. These components (and many other kinds as well) can be represented as program transformations[2,5,26]. A transformation matches against some section of program text (or more usually its parse tree) and replaces it by a new section of program text (or parse tree). A typical transformation has three parts. It has a pattern which matches against the program in order to determine where to apply the transformation. It has a set of logical applicability conditions which further restrict the places where the transformation can be applied. Finally, it has a (usually procedural) action which creates the new program section based on the old section. Note that when applied to small localized sections of a program, program transformations are very much the same as macros.

An important aspect of program transformations is the idea of a *wide spectrum language*. In contrast to ordinary high-level languages, wide spectrum languages contain syntactic and semantic extensions which are not directly executable. In some cases these higher level constructs have a semantics independent of the transformation system, but often they are defined only in terms of the transformations which convert them into executable constructs.

The most interesting contribution of transformations is that they view program construction as a *process*. Rather than viewing a program solely as a static artifact which may be decomposed into components the way a house is made up of a floor, roof and walls, transformations view a program as evolving through a series of construction steps which utilize components which may not be visible in the final program, just as the construction of a house requires the use of scaffolding and other temporary structures. This point of view enables transformations to express components such as *move invariant* which are common steps in the construction of a program rather than common steps in the execution of a program.

Another important aspect of transformations is that they can be combined in a way which is quite different from the other formalisms. As mentioned above, many simple transformations are basically just macros which specify how to implement particular high-level constructs in a wide spectrum language. These transformations are only triggered when instances of their associated high-level constructs appear; thus they only operate where they are explicitly requested and combine in much the same way as macros.

However, other transformations are much less localized in the way the operate. For example, a transformation representing *move invariant* would have applicability conditions (e.g., that the expression is invariant) which must look at large parts of the program. In addition, such transformations are not intended to be applied only when explicitly requested by the user. Rather, they are intended to be used whenever they become applicable for any reason. This makes powerful synergistic interaction between transformations possible.

Unfortunately, if transformations are allowed to contain arbitrary computation in their actions, they have the same difficulty with regard to semantic soundness and convenient combinability that macros have. The transformation writer has to take great care in order to insure that the interaction between transformations will in fact be synergistic rather than antagonistic. In order to have a semantic basis, transformations must include a logical description of what the transformation is doing. One important way that this has been done is to focus on transformations which are correctness preserving—ones which, from a logical perspective, do nothing.

A number of experimental systems have been developed which demonstrate that transformations are machine manipulable[3,6,7]. All of these systems support the automatic application of transformations. Some of them go beyond this to attack the harder problem of automatically selecting the transformations to apply.

A difficulty with transformations is that, as generally supported, they are very much programming language dependent. This not only limits the portability of components represented as transformations, it also limits the way transformations can be stated by requiring that every intermediate state of a program being transformed has to fit into the syntax of the programming language. One way to alleviate these problems would be to apply transformations to a programming language independent representation such as flowcharts.

The Plan Calculus

As part of the Programmer's Apprentice project, a formalism has been developed which seeks to satisfy all of the desiderata by combining several of the techniques discussed above. In this formalism (called the Plan Calculus[21]) components are represented as *plans*. Plans contain three kinds of information: *plan diagrams*, *logical annotation*, and *overlays*.

A plan diagram contains information about the algorithmic aspects of a plan. In order to achieve language independence, plan diagrams are represented as hierarchical, data flow schemas. In a plan diagram, computations are represented as boxes with input and output ports while control flow and data flow are both represented using arcs between ports. In addition, plan diagrams are hierarchical—a box in a diagram can contain an entire sub-diagram. Further, the diagrams are schematic—they can contain empty boxes (called *roles*) which are to be filled in later.

As an example of a plan diagram, Figure 2 shows the algorithmic part of a plan for computing the absolute value of a number. In the figure, data flow arcs are drawn as solid lines and control flow arcs as hatched lines. The diagram is composed of an operation box (`action`) whose output is the negation of its input, a test box (`if`) which splits control based on whether or not its input is negative, and a "join" box (`end`) which rejoins the control split by the test. The output of the join is determined by the control flow path which is used to enter it.

The non-algorithmic aspects of a plan are represented using predicate calculus assertions. These assertions are attached as annotations on the plan diagram. Each box in a plan diagram is annotated with a set of preconditions and postconditions. In addition, logical constraints between roles are used to limit the way in which the roles can to be filled in. Finally, dependency links record a summary of a proof that the specifications of the plan as a whole follow from the specifications of the inner boxes and the way these boxes are connected. Components such as *master file system* and *deadlock free* which have little or no algorithmic aspect are represented by plans which consist almost entirely of predicate calculus assertions with little or no diagrammatic information.

In order to unify the concept of a plan for an algorithm with the concept of a plan for a data structure, the basic flowchart-like ideas behind plan diagrams are extended so that plan diagrams can contain parts which correspond to data objects as well as sub-computations. Data parts can be left unspecified as data roles and data parts can be annotated with specifications, constraints, and dependencies. Given these extensions, plans are capable of representing the same kinds of information as data abstraction mechanisms. For example, the plan for *stack* consists of a number of logically interrelated plan diagrams, one of which represents the stack data object and the rest of which represent the operations on a stack.

The transformational aspects of a plan are represented as *overlays*. An overlay is a mapping between two plans. It specifies a set of correspondences between the roles of the plans. Overlays are similar to transformations in which both the left and right hand side are plans. However, overlays differ from program transformations in two ways: they are bidirectional and their actions are declarative as opposed to procedural. The fact that overlays are bidirectional means that, like grammar rules, they can be used for both analysis and synthesis. The fact that overlays are totally declarative gives them a firm semantic basis and makes it easier to reason about them as opposed to merely using them.

Figure 3 shows an example of an overlay which specifies how to transform a tail recursive

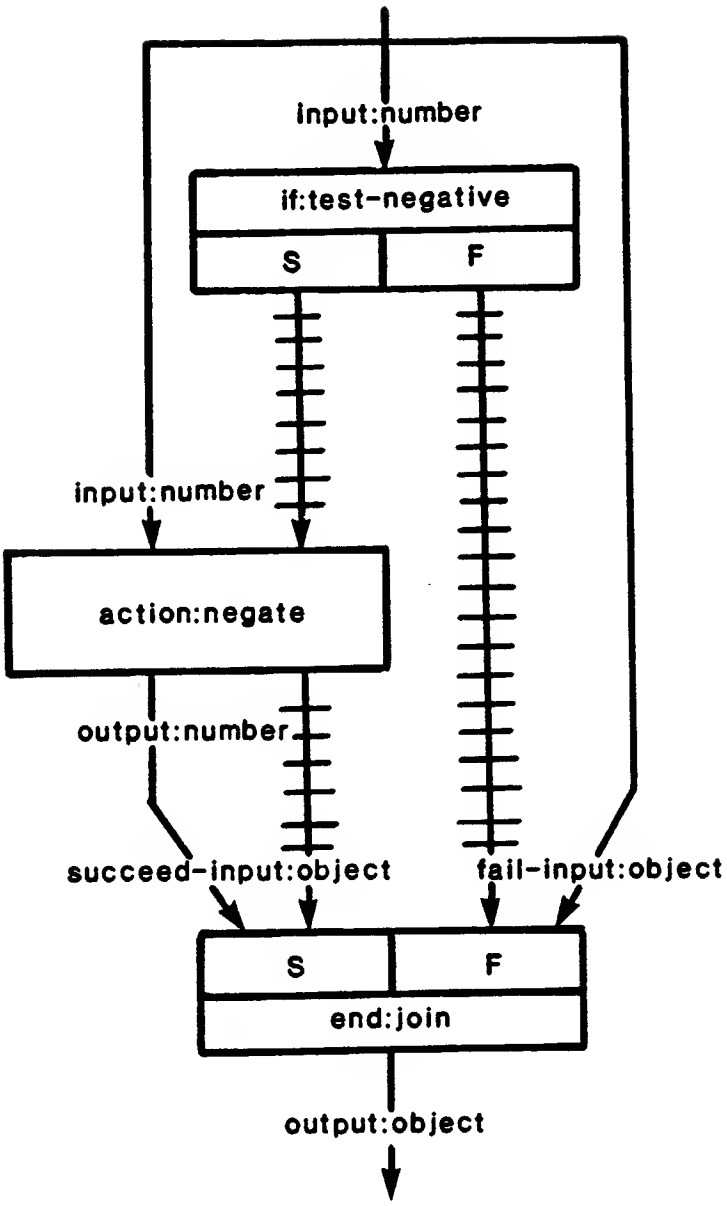

Figure 2: An Example Plan.

program which accumulates a value "on the way down" into a recursive program which accumulates the same result "on the way up" and vice versa. This overlay captures the commonality between the programs `SUM_UP` and `SUM_DOWN` shown below.

```
SUM_DOWN(L) = SUM_DOWN2(L,0);
SUM_DOWN2(L,S) = if EMPTY(L) then S else SUM_DOWN2(TAIL(L),S+HEAD(L));

SUM_UP(L) = if EMPTY(L) then 0 else SUM_UP(TAIL(L))+HEAD(L);
```

The plan diagrams in Figure 3 are drawn in the same way as in Figure 2. However, three new features are shown. Dashed boxes are used to group diagrams hierarchically. looping lines are used to indicated that a dashed box in a diagram is identical to the dashed box

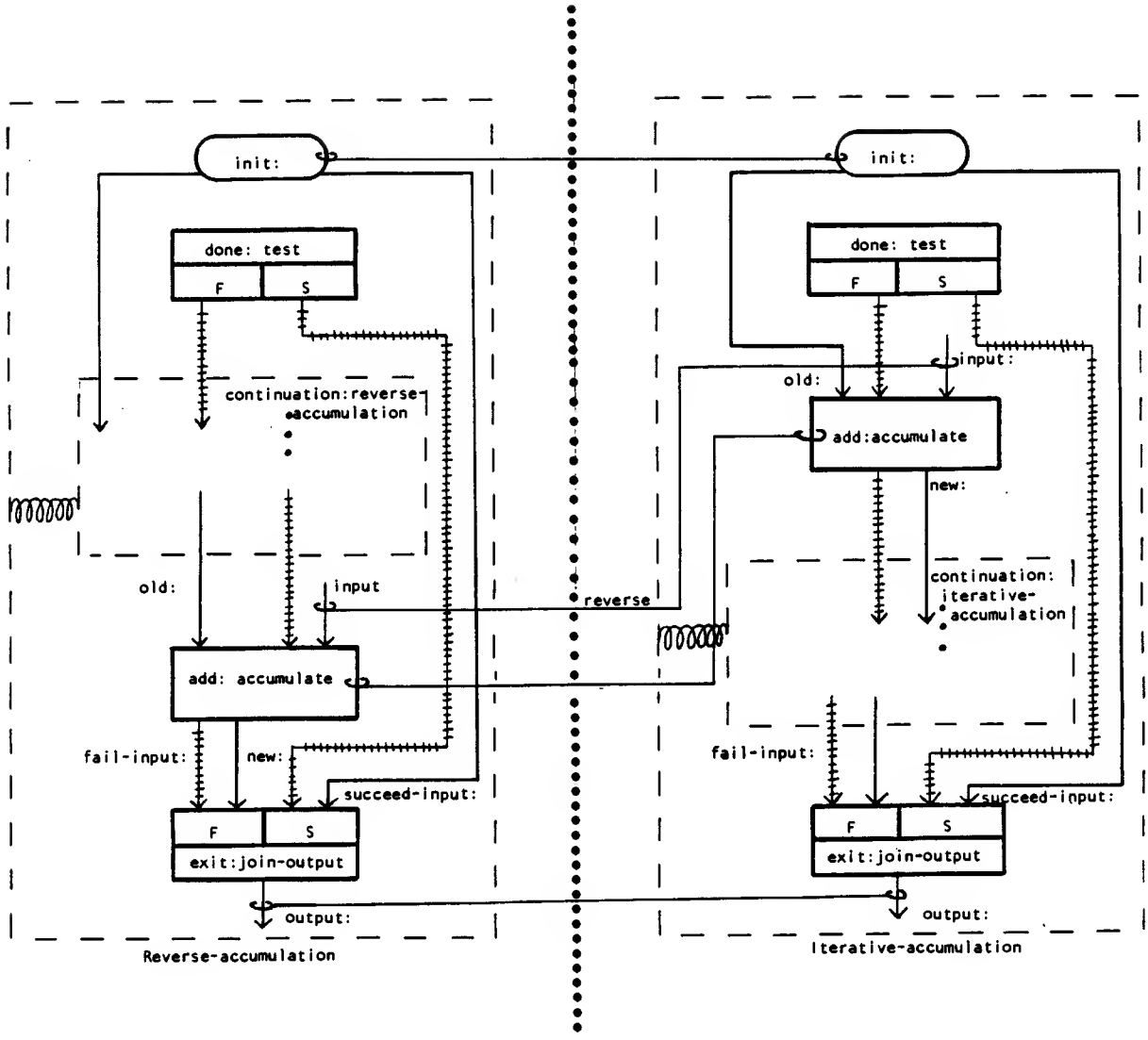

Figure 3: An Example Overlay.

containing it. (The right and left diagrams in the figure are both infinite.) Finally, hooked lines are used to indicate the overlay relationship between the two diagrams.

The plan on the left side of Figure 3 represents accumulation of a result "on the way up" while the plan on the right hand side of the figure represents tail recursive accumulation. The four hooked lines specify correspondences. Unlabeled correspondences are equalities. The initialization (`init`), accumulation (`add`), and output values in the two plans correspond directly. The correspondence which is labeled with `reverse` is more complex. It specifies that the order of the input elements on the right hand side is the reverse of the order of the input elements on the left hand side. (The storage of elements on the function invocation stack performs the reversal.) A logical assertion (not shown in the figure) indicates that the final output in the two plans is not equal unless the `add` operation is associative.

In order to give plan diagrams a precise definition, each aspect of plan diagrams is defined in terms of a version of the situational calculus[11]. Manna and Waldinger have used the

situational calculus in a similar way in order to specify certain problematic features of programming languages[19].

Since the situational calculus is essentially just predicate calculus with some conventions applied, a plan diagram can be viewed as the abbreviation for a set of predicate calculus assertions. Given that the logical annotation in a plan consists directly of predicate calculus assertions and overlays are just mappings between plans, this implies that everything in a plan can be reduced to a set of logical assertions.

In summary, the Plan Calculus combines the expressiveness of hierarchical data flow schemas, logical formalisms, and transformations. Given that each of these mechanisms is programming language independent, the Plan Calculus is language independent as well. The fact that plans can, in principle, be translated into predicate calculus gives the Plan Calculus the same kind of rigorous semantic foundation that any logical formalism has. In addition, the combination of two plans amounts semantically to the union of axioms and is therefore convenient from a theoretical viewpoint. Finally, since any plan can be represented as a set of assertions, the Plan Calculus is, in principle, machine manipulable by an automatic reasoning system. The only question which remains unanswered is whether or not the Plan Calculus is machine manipulable in practice as opposed to merely in principle.

A Hybrid Reasoning System For Plans

The weakness of current automatic reasoning systems implies that general-purpose deduction cannot be used by itself to manipulate a complex representation such as the Plan Calculus. Experimentation has shown that a hybrid system which combines special-purpose techniques with general-purpose logical reasoning is required.

Special-purpose representations and algorithms are essential in order to avoid the combinatorial explosions that typically occur in general-purpose reasoning systems. On the other hand, logic-based reasoning is very valuable when used, under strict control, as the "glue" between inferences made in different special-purpose representations.

A hybrid reasoning system, called CAKE[22], is being implemented, which is tailored specifically for reasoning about plans. Figure 4 shows the architecture of CAKE. The bottom layers of CAKE support general-purpose logical reasoning while the top two layers support special-purpose reasoning about plans.

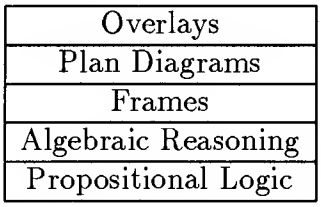

Figure 4: The layers of CAKE.

Although the information in a plan which is represented by means of plan diagrams and overlays could be converted into logical assertions, it is for the most part not converted. Rather, this information is represented in terms of graphical data structures which can be manipulated by means of special-purpose procedures. For example, the combination of plan

diagrams can be performed by substituting one diagram into the other. General-purpose reasoning is necessary only for the much simpler task of checking whether the relevant preconditions and constraints permit the substitution to take place.

At the current time, the three lowest layers of CAKE have been completed. Figure 5 is a short transcript illustrating some of the facilities provided. (Line numbers in the following discussion refer to Figure 5.)

The propositional layer of CAKE provides three principal facilities. First, it automatically performs simple "one-step" deductions (lines 1–3). Second, it acts as a recording medium for dependencies, and thus supports explanation (line 3) and retraction (lines 4–5). Third, this layer detects contradictions (lines 6–7). Contradictions are represented explicitly in such a way that reasoning can continue with other information not involved in the contradiction. This is important for allowing a user to postpone dealing with problems.

The algebraic layer of CAKE is composed of special-purpose decision procedures for congruence closure, common algebraic properties of operators (i.e., commutativity, associativity, and transitivity), and the algebra of sets. The congruence closure algorithm in this layer determines whether or not terms are equal by substitution of equal subterms (lines 8–9). The decision procedure for transitivity (lines 10–12) determines when elements of a binary relation follow by transitivity from other elements. The algebra of sets (lines 13–15) involves the theory of membership, subset, union, intersection and complements. (The propositional layer and the congruence closure algorithm of the algebraic layer are derived from McAllester's Reasoning Utility Package[20].)

The frames layer, which is built using facilities from the layers below, supports the conventional frame notions of inheritance (`:Specializes` in line 16), slots (`:Roles` in lines 17–19), and instances (line 20). A notable feature of CAKE's frame system is that constraints are implemented in a general way. For example, the definition of an `Interface` (line 19) has constraints between the roles of the frames filling its roles. When an instance of this frame is created (line 20) and a particular value (`777777`) is put into one of its roles (line 21), the same value can be retrieved from the other constrained role (line 22). This propagation is not achieved by *ad hoc* procedures, but by the operation of the underlying logical reasoning system, including dependencies (line 23).

It should be realized that CAKE's general-purpose reasoning capabilities are relatively weak. Therefore, the practicality of CAKE as a tool for manipulating the Plan Calculus depends on the fact that most of the information in a plan is represented diagrammatically as opposed to logically. As a result, it is unlikely that CAKE will be able to deal effectively with components such as *deadlock free* which have little or no algorithmic aspect and which require complex reasoning. However, it should be able to effectively manipulate all of the other kinds of components sited as examples above.

```
 1>  (Assertq P)
 2>  (Assertq (Implies P Q))
 3>  (Whyq Q)
     Q is TRUE by Modus Ponens from:
       1. (IMPLIES P Q) is TRUE as a premise.
       2. P is TRUE as a premise.
 4>  (Retractq P)
 5>  (Whyq Q)
     I don't know whether or not Q is true.
 6>  (Assertq (And P (Not Q)))
     >>Contradiction: There is a conflict between the premises:
         1. (AND P (NOT Q)) is TRUE.
         2. (IMPLIES P Q) is TRUE.
     s-A, Resume:   Ignore this contradiction.
     s-B:           Retract one of the premises.
 7>  s-B Retract one of the premises.
     Premise to retract: 1
     Retracting (AND P (NOT Q)) being TRUE...
     #<Node (AND P (NOT Q)): False>
 8>  (Assertq (= I J))
 9>  (Whyq (= (F I) (F J)))
     (= (F I) (F J)) is TRUE by Equality from:
       1. (= I J) is TRUE as a premise.
10>  (Assertq (Transitive R))
11>  (Assertq (R W X)) (Assertq (R X Y)) (Assertq (R Y Z))
12>  (Whyq (R W Z))
     (R W Z) is TRUE by Transitivity from:
       1. (R W X) is TRUE as a premise.
       2. (R X Y) is TRUE as a premise.
       3. (R Y Z) is TRUE as a premise.
       4. (TRANSITIVE R) is TRUE as a premise.
13>  (Assertq (Subset A B))
14>  (Assertq (Member X A))
15>  (Whyq (Member X B))
     (MEMBER X B) is TRUE by Subsumption from:
       1. (SUBSET A B) is TRUE as a premise.
       2. (MEMBER X A) is TRUE as a premise.
16>  (Deftype Address (:Specializes Number))
17>  (Deframe Interrupt (:Roles (Location Address) Program))
18>  (Deframe Device (:Roles (Transmit Address) (Receive Address)))
19>  (Deframe Interface
       (:Roles (Target Device) (From Interrupt) (To Interrupt))
       (:Constraints (= (Location ?From) (Receive ?Target))
                     (= (Location ?To) (Transmit ?Target))))
20>  (FInstantiate 'Interface :Name 'K7)
21>  (FPut (>> 'K7 'Target 'Receive) 777777)
22>  (FGet (>> 'K7 'From 'Location))
     777777
23>  (Why ...)
     (= 777777 (LOCATION (FROM K7))) is TRUE by Equality from:
       1. (= (LOCATION (FROM K7))
             (RECEIVE (TARGET K7))) is TRUE.
       2. (= (RECEIVE (TARGET K7)) 777777) is TRUE as a premise.
```

Figure 5: A transcript showing CAKE in action.

The Knowledge-Based Editor in Emacs

The Knowledge-Based Editor in Emacs (KBEmacs) is the current demonstration system implemented as part of the Programmer's Apprentice project. KBEmacs [27] is a program editor which makes it possible to construct programs rapidly and reliably by combining algorithmic components represented as plans. From the point of view of the current discussion, the key feature of KBEmacs is that it demonstrates the machine manipulability of plan diagrams.

KBEmacs is implemented on the Symbolics Lisp Machine [31]. Figure 6 shows the architecture of the system. KBEmacs maintains two representations for the program being worked on: program text and a plan.

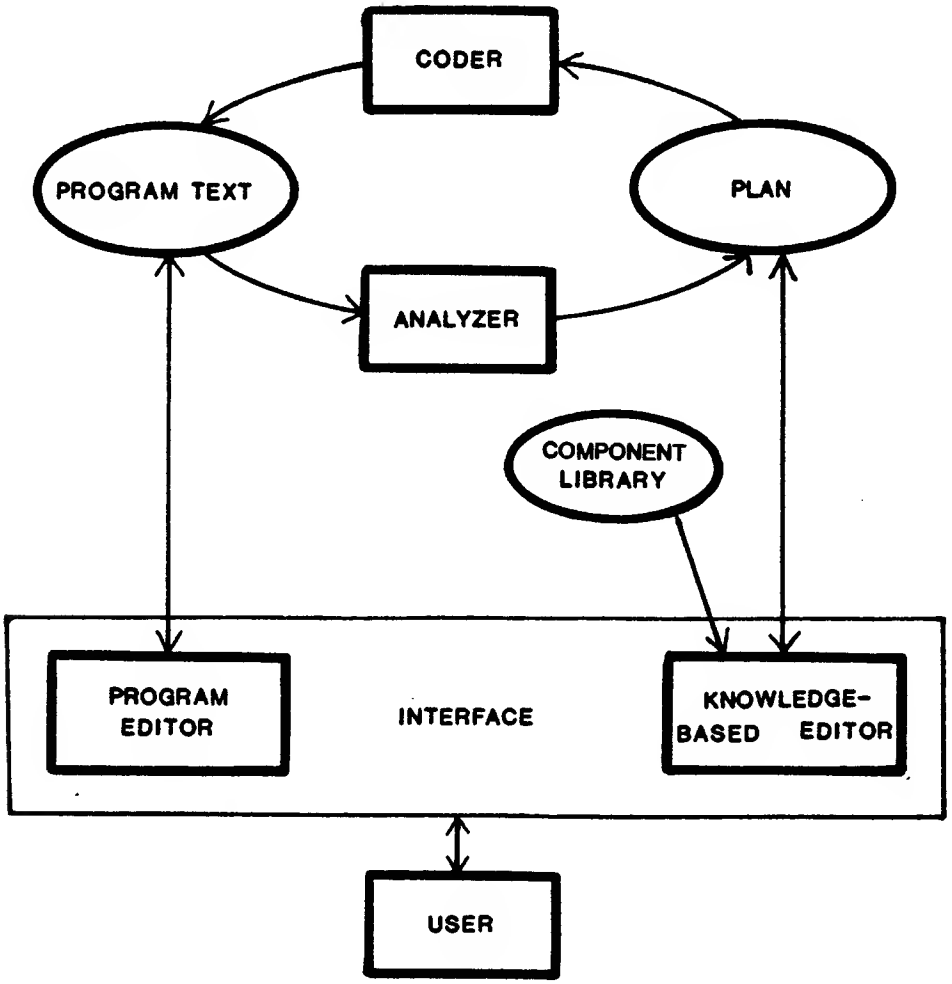

Figure 6: The architecture of KBEmacs.

KBEmacs is based on a simple, early plan representation. (While KBEmacs was under construction, this simple plan representation evolved into the Plan Calculus.) The simple representation corresponds to the plan diagram portion of the Plan Calculus with the addition of support for procedural constraints.

At any moment, the programmer can either modify the program text or the plan. In order to modify the program text, the programmer can use the standard Emacs-style Lisp

Machine editor. This editor supports both text-based and syntax-based program editing.

In order to modify the plan, the programmer can use the *knowledge-based editor* module. This module supports several commands for instantiating and combining components. Each command is supported by a special-purpose procedure which operates directly on the plan representation. The knowledge-based editor also provides support for maintaining the consistency of procedurally represented constraints. However, it does not have any general-purpose reasoning abilities.

The components themselves are represented as plans and stored in the *component library*. New components can be defined by the user by using a programming-language-like representation.

An interface unifies ordinary program editing and knowledge-based editing so that they can both be conveniently accessed through the standard Lisp Machine editor. The knowledge-based commands are supported as an extension of the standard editor command set and the results of these commands are communicated to the programmer by altering the program text in the editor buffer. The effect is the same as if a human assistant were sitting at the editor modifying the text under the direction of the programmer.

Whenever the plan is modified, the *coder* module is used to create new program text. The coder operates in three steps. First, it examines the plan in order to determine how the control flow should be implemented—i.e., determining where conditional and looping constructs should be used. Second, it determines how the data flow should be implemented—i.e., when variables should be used and what names should be used for them. Third, it constructs program text based on the decisions above and then applies various transformations in order to improve the readability of the result. The complexity of the coder stems not from the need to create correct code (this is relatively easy) but from the need to create aesthetic code.

Whenever the program text is modified, the *analyzer* module is used to create a new plan. The analyzer is similar to the front end of an optimizing compiler. It operates on a program in four stages. First, the program is parsed. Second, macro expansion is used to express the various constructs of the language in terms of primitive operations. For example, all control constructs are expanded into `gotos`. Third, the resulting intermediate form is processed to determine the basic functions called by the program, the roles in the program, and the data flow and control flow between them. This results in the construction of a simple plan. Fourth, the plan is analyzed in order to determine the hierarchical structure of the program.

Scenario

In order to give a feeling for the capabilities of KBEmacs, this section presents a condensed summary of the scenario in [27]. In that scenario, a programmer uses KBEmacs to construct an Ada program in the domain of business data processing. It is assumed that there is a data base which contains information about various machines (referred to as *units*) sold by a company and about the repairs performed on each of these units. In the scenario, the programmer constructs a program called `UNIT_REPAIR_REPORT` which prints out a report of all of the repairs performed on a given unit. The directions in Figure 7 might be given to a human assistant who was asked to write this program.

A key feature of these directions is that they refer to a significant amount of knowledge

```
Define a simple report program UNIT_REPAIR_REPORT.  Enumerate the
chain of repairs associated with a unit record, printing each one.
Query the user for the key (UNIT_KEY) of the unit record to start
from.  Print the title ("Report of Repairs on Unit " & UNIT_KEY).
Do not print a summary.
```

Figure 7: Hypothetical directions for a human assistant.

that the assistant is assumed to possess. First, they refer to a number of standard programming algorithms— i.e., "simple report", "enumerating the records in a chain", "querying the user for a key". Second, they assume that the assistant understands the structure of the data base of units and repairs. Another feature of the directions is that, given that the assistant has a precise understanding of the algorithms to be used and of the data base, little is left to the assistant's imagination. Essentially every detail of the algorithm is spelled out, including the exact Ada code to use when printing the title.

Using KBEmacs, the commands shown in Figure 8 can be used to construct the program UNIT_REPAIR_REPORT. The Ada program which results from these commands is shown in Figure 9.

```
Define a simple_report procedure UNIT_REPAIR_REPORT.
Fill the enumerator with a chain_enumeration of UNITS and REPAIRS.
Fill the main_file_key with a query_user_for_key of UNITS.
Fill the title with ("Report of Repairs on Unit " & UNIT_KEY).
Remove the summary.
```

Figure 8: KBEmacs commands.

A key feature of the commands in Figure 8 is that they refer to a number of components known to KBEmacs—i.e., *simple-report*, *chain-enumeration*, and *query-user-for-key*. In addition, they assume an understanding of the structure of the data base. (KBEmacs understands the data base because it can understand the Ada package which defines the data base.) The "Fill" commands specify how to fill in the roles of the plan for *simple-report*.

Without discussing either the commands or the program produced in any detail, two important observations can be made. First, the commands used are very similar to the hypothetical directions for a human assistant. Second, a set of 5 commands produces a 55 line program. (The program would be even longer if it did not make extensive use of data declarations and functions defined in the packages FUNCTIONS and MAINTENANCE_FILES.)

The KBEmacs commands and the hypothetical directions differ in grammatical form, but not in semantic content. This is not surprising in light of the fact that the hypothesized commands were in actuality created by restating the knowledge-based commands in more free flowing English.

The purpose of this translation was to demonstrate that although the KBEmacs commands may be syntactically awkward, they are not semantically awkward. The commands are neither redundant nor overly detailed. They specify only the basic design decisions which underly the program. There is no reason to believe that any automatic system (or

```
with CALENDAR, FUNCTIONS, MAINTENANCE_FILES, TEXT_IO;
use CALENDAR, FUNCTIONS, MAINTENANCE_FILES, TEXT_IO;
procedure UNIT_REPAIR_REPORT is
   use DEFECT_IO, REPAIR_IO, UNIT_IO, INT_IO;
   CURRENT_DATE: constant STRING := FORMAT_DATE(CLOCK);
   DEFECT: DEFECT_TYPE;
   REPAIR: REPAIR_TYPE;
   REPAIR_INDEX: REPAIR_INDEX_TYPE;
   REPORT: TEXT_IO.FILE_TYPE;
   TITLE: STRING(1..33);
   UNIT: UNIT_TYPE;
   UNIT_KEY: UNIT_KEY_TYPE;
   procedure CLEAN_UP is
      begin
         SET_OUTPUT(STANDARD_OUTPUT);
         CLOSE(DEFECTS); CLOSE(REPAIRS); CLOSE(UNITS); CLOSE(REPORT);
      exception
         when STATUS_ERROR => return;
      end CLEAN_UP;
begin
   OPEN(DEFECTS, IN_FILE, DEFECTS_NAME); OPEN(REPAIRS, IN_FILE, REPAIRS_NAME);
   OPEN(UNITS, IN_FILE, UNITS_NAME); CREATE(REPORT, OUT_FILE, "report.txt");
   loop
      begin
         NEW_LINE; PUT("Enter UNIT Key: "); GET(UNIT_KEY);
         READ(UNITS, UNIT, UNIT_KEY);
         exit;
      exception
         when END_ERROR => PUT("Invalid UNIT Key"); NEW_LINE;
      end;
   end loop;
   TITLE := "Report of Repairs on Unit " & UNIT_KEY;
   SET_OUTPUT(REPORT);
   NEW_LINE(4); SET_COL(20); PUT(CURRENT_DATE);
   NEW_LINE(2); SET_COL(13); PUT(TITLE); NEW_LINE(60);
   READ(UNITS, UNIT, UNIT_KEY); REPAIR_INDEX := UNIT.REPAIR;
   while not NULL_INDEX(REPAIR_INDEX) loop
      READ(REPAIRS, REPAIR, REPAIR_INDEX);
      if LINE > 64 then
         NEW_PAGE; NEW_LINE; PUT("Page: "); PUT(INTEGER(PAGE-1), 3);
         SET_COL(13); PUT(TITLE); SET_COL(61); PUT(CURRENT_DATE); NEW_LINE(2);
         PUT("   Date     Defect   Description/Comment"); NEW_LINE(2);
      end if;
      READ(DEFECTS, DEFECT, REPAIR.DEFECT);
      PUT(FORMAT_DATE(REPAIR.DATE)); SET_COL(13); PUT(REPAIR.DEFECT);
      SET_COL(20); PUT(DEFECT.NAME); NEW_LINE;
      SET_COL(22); PUT(REPAIR.COMMENT); NEW_LINE;
      REPAIR_INDEX := REPAIR.NEXT;
   end loop;
   CLEAN_UP;
exception
   when DEVICE_ERROR | END_ERROR | NAME_ERROR | STATUS_ERROR =>
      CLEAN_UP; PUT("Data Base Inconsistent");
   when others => CLEAN_UP; raise;
end UNIT_REPAIR_REPORT;
```

Figure 9: The Ada program UNIT_REPAIR_REPORT.

for that matter a person) could be told how to construct the program UNIT_REPAIR_REPORT without being told at least most of the information in the commands shown.

The leverage that KBEmacs applies to the program construction task is illustrated by the order of magnitude difference between the size of the set of commands and the size of the program. A given programmer seems to be able to produce more or less a constant number of lines of code per day independent of the programming language being used. As a result, there is reason to believe that the order of magnitude size reduction provided by the KBEmacs commands would translate into an order of magnitude reduction in the time required to construct the program. It should be noted that since program construction is only a small part (around 10%) of the programming life cycle, this does not translate into an order of magnitude savings in the life cycle as a whole.

Another important advantage of KBEmacs is that using standard components (such as *simple-report* and *chain-enumeration*) enhances the reliability of the programs produced. Since the components known to KBEmacs are intended to be used many times, it is economically justifiable to lavish a great deal of time on them in order to ensure that they are general-purpose and bug free. This reliability is inherited by the programs which use the standard algorithms.

When using an ordinary program editor, programmers typically make two kinds of errors: picking the wrong algorithms to use and incorrectly instantiating these algorithms (i.e., combining the algorithms together and rendering them as appropriate program code). KBEmacs eliminates the second kind of error.

Combining Components

As an illustration of the way KBEmacs supports the combination of components represented as plans, Figures 10 and 11 show two steps in the creation of the program in Figure 9.

Figure 10 shows the program text which is produced by KBEmacs after the first command in Figure 8. This code comes almost entirely from the component *simple-report*. This component specifies the high-level structure of a simple reporting program. The component has several roles which are are represented using the notation {...}.

The *title* role is printed on a title page and, along with the page number, at the top of each succeeding page of the report. The *enumerator* enumerates the elements of some aggregate data structure. The *print-item* is used to print out information about each of the enumerated elements. The *column-headings* are printed at the top of each page of the report in order to explain the output of the *print-item*. The *summary* prints out some summary information at the end of the report. The *print-item*, *column-headings*, and *summary* are all computations which side-effect the report file (which is used as the value of STANDARD_OUTPUT) by sending output to it.

The *enumerator* is different from the other roles in that it is *compound*—consisting of four sub-roles distributed through the program. Compound roles are used when the syntactic limitations of a programming language prevent a logical unit from be expressed as a syntactic unit. In order to facilitate component combination, the *enumerator* is represented as a single box in the plan for *simple-report*.

The *input structure* of the *enumerator* corresponds to the aggregate structure to be enumerated. The *empty-test* determines whether all of the elements in the aggregate structure

```
with CALENDAR, FUNCTIONS, TEXT_IO;
use CALENDAR, FUNCTIONS, TEXT_IO;
procedure UNIT_REPAIR_REPORT is
   use INT_IO;
   CURRENT_DATE: constant STRING := FORMAT_DATE(CLOCK);
   DATA: {};
   REPORT: TEXT_IO.FILE_TYPE;
   TITLE: STRING(1..{});
   procedure CLEAN_UP is
      begin
         SET_OUTPUT(STANDARD_OUTPUT);
         CLOSE(REPORT);
      exception
         when STATUS_ERROR => return;
      end CLEAN_UP;
begin
   CREATE(REPORT, OUT_FILE, "report.txt");
   TITLE := {the title};
   SET_OUTPUT(REPORT);
   NEW_LINE(4); SET_COL(20); PUT(CURRENT_DATE); NEW_LINE(2);
   SET_COL(13); PUT(TITLE); NEW_LINE(60);
   DATA := {the input structure of the enumerator};
   while not {the empty_test of the enumerator}(DATA) loop
      if LINE > 64 then
         NEW_PAGE; NEW_LINE; PUT("Page: "); PUT(INTEGER(PAGE-1), 3);
         SET_COL(13); PUT(TITLE); SET_COL(61); PUT(CURRENT_DATE); NEW_LINE(2);
         {the column_headings}({CURRENT_OUTPUT, modified});
      end if;
      {the print_item}({CURRENT_OUTPUT, modified},
                      {the element_accessor of the enumerator}(DATA));
      DATA := {the step of the enumerator}(DATA);
   end loop;
   {the summary}({CURRENT_OUTPUT, modified});
   CLEAN_UP;
exception
   when DEVICE_ERROR | END_ERROR | NAME_ERROR | STATUS_ERROR =>
      CLEAN_UP; PUT("Data Base Inconsistent");
   when others => CLEAN_UP; raise;
end UNIT_REPAIR_REPORT;
```

Figure 10: Results of "`Define a simple_report program UNIT_REPAIR_REPORT`".

have been enumerated and therefore whether the enumeration should be terminated. The *element-accessor* accesses the current element in the aggregate. The *step* steps from one element of the aggregate structure to the next.

There is an additional role of the component *simple-report* which is of particular importance. This role is called the *line-limit* and is used to determine when a page break should be inserted in the report. The presence of this role is not obvious in Figure 10 because it has already been filled in with the default value 64. This value was generated by a constraint.

The most interesting feature of the component *simple-report* is the fact that it contains the constraints shown below.

```
constraints
   DERIVED(the line_limit,
           66-SIZE_IN_LINES(the print_item)
             -SIZE_IN_LINES(the summary));
   DEFAULT(the print_item,
           CORRESPONDING_PRINTING(the enumerator));
   DEFAULT(the column_headings,
           CORRESPONDING_HEADINGS(the print_item));
end constraints;
```

The first constraint specifies that the *line-limit* is constrained to be 66 minus the number of lines printed by the *print-item* and the number of lines printed by the *summary*. Under the assumption that there is room for 66 lines on a page of output, the constraint guarantees that, whenever the line number is less than or equal to the *line-limit*, there will be room for both the *print-item* and the *summary* to be printed on the current page. Because the *line-limit* role is derived by this constraint the programmer never has to fill it in explicitly, and the role will be automatically updated whenever either the *print-item* or the *summary* is changed.

The other two constraints specify default values for the *print-item* and *column-headings* roles. The function `CORRESPONDING_PRINTING` determines what should be used to fill in the *print-item* role based on the type of object which is being enumerated. The function `CORRESPONDING_HEADINGS` determines what headings should be used based on the way the *print-item* is filled in.

The functions `CORRESPONDING_PRINTING` and `CORRESPONDING_HEADINGS` operate in one of two modes. In general, components will have been defined which specify how to print out a given type of object in a report, and how to print the corresponding headings. If this is the case, then the functions `CORRESPONDING_PRINTING` and `CORRESPONDING_HEADINGS` merely retrieve the appropriate components. However, if there are no such components, then the functions `CORRESPONDING_PRINTING` and `CORRESPONDING_HEADINGS` use a simple program generator in order to construct appropriate code based on the definition of the type of object in question.

Figure 11 shows the program code produced by KBEmacs after the second command in Figure 8. Change indicators in the left margin of the figure indicate the lines where something has changed in comparison with Figure 10. These changes, which result from combining the components *simple-report* and *chain-enumeration*, are spread throughout the program.

The lines marked with `1` indicate changes which come directly from the component *chain-enumeration*. This includes code for each of the sub-roles of the *enumerator*.

Once the *enumerator* has been specified, the constraints (described above) in the component *simple-report* fill in most of the rest of the program `UNIT_REPAIR_REPORT`. It is assumed that two components (*print-repair* and *print-repair-headings*) exist which specify how to print a repair record and the associated column headings. (The lines marked with `2` and `3` come from these two components respectively.)

When the *print-item* is filled in, the *line-limit* is changed from `64` to `63` since the code used to fill the *print-item* generates two lines of output, whereas the default assumption used by the constraint function `SIZE_IN_LINES` is that only one line of output will be produced. (This change is marked with a `4`.)

The automatic updating of the *line-limit* role is a good example of the way KBEmacs can

```
5 with CALENDAR, FUNCTIONS, MAINTENANCE_FILES, TEXT_IO;
5 use CALENDAR, FUNCTIONS, MAINTENANCE_FILES, TEXT_IO;
  procedure UNIT_REPAIR_REPORT is
5    use DEFECT_IO, REPAIR_IO, UNIT_IO, INT_IO;
     CURRENT_DATE: constant STRING := FORMAT_DATE(CLOCK);
5    DEFECT: DEFECT_TYPE;
5    REPAIR: REPAIR_TYPE;
5    REPAIR_INDEX: REPAIR_INDEX_TYPE;
     REPORT: TEXT_IO.FILE_TYPE;
     TITLE: STRING(1..{});
5    UNIT: UNIT_TYPE;
     procedure CLEAN_UP is
        begin
           SET_OUTPUT(STANDARD_OUTPUT);
5          CLOSE(DEFECTS); CLOSE(REPAIRS); CLOSE(UNITS); CLOSE(REPORT);
        exception
           when STATUS_ERROR => return;
        end CLEAN_UP;
  begin
2    OPEN(DEFECTS, IN_FILE, DEFECTS_NAME);
1    OPEN(REPAIRS, IN_FILE, REPAIRS_NAME); OPEN(UNITS, IN_FILE, UNITS_NAME);
     CREATE(REPORT, OUT_FILE, "report.txt");
     TITLE := {the title};
     SET_OUTPUT(REPORT);
     NEW_LINE(4); SET_COL(20); PUT(CURRENT_DATE); NEW_LINE(2);
     SET_COL(13); PUT(TITLE); NEW_LINE(60);
1    READ(UNITS, UNIT, {the main_file_key});
1    REPAIR_INDEX := UNIT.REPAIR;
1    while not NULL_INDEX(REPAIR_INDEX) loop
1       READ(REPAIRS, REPAIR, REPAIR_INDEX);
4       if LINE > 63 then
           NEW_PAGE; NEW_LINE; PUT("Page: "); PUT(INTEGER(PAGE-1), 3);
           SET_COL(13); PUT(TITLE); SET_COL(61); PUT(CURRENT_DATE); NEW_LINE(2);
3          PUT("   Date     Defect   Description/Comment"); NEW_LINE(2);
        end if;
2       READ(DEFECTS, DEFECT, REPAIR.DEFECT);
2       PUT(FORMAT_DATE(REPAIR.DATE)); SET_COL(13); PUT(REPAIR.DEFECT);
2       SET_COL(20); PUT(DEFECT.NAME); NEW_LINE;
2       SET_COL(22); PUT(REPAIR.COMMENT); NEW_LINE;
1       REPAIR_INDEX := REPAIR.PREVIOUS;
     end loop;
     {the summary}({CURRENT_OUTPUT, modified});
     CLEAN_UP;
  exception
     when DEVICE_ERROR | END_ERROR | NAME_ERROR | STATUS_ERROR =>
        CLEAN_UP; PUT("Data Base Inconsistent");
     when others => CLEAN_UP; raise;
  end UNIT_REPAIR_REPORT;
```

Figure 11: Results of "`Fill the enumerator with a chain_enumeration of ...`".

enhance program reliability. The main leverage KBEmacs applies to the reliability problem is that each component is internally consistent. The use of constraints can help maintain this consistency.

If KBEmacs had not updated the *line-limit* role, the programmer might not have realized that it needed to be updated. The bug which would result, though minor, would have the pernicious quality of being rather hard to detect during program testing since the bug only manifests itself when the program attempts to print the summary as the very bottom of a page.

A final thing to note about the code in Figure 11 is that a number of variable declarations and the like have been added to the program (the lines marked with 5). This is an example of the fact that KBEmacs can automatically take care of several kinds of programming details. It is interesting to note that the data types in these declarations are not specified as part of the components used. Rather, KBEmacs computes what data types should be used based on the definitions of the relevant files and the specifications for the procedures which operate on the variables.

After specifying the *enumerator* in Figure 11 the only roles which are left unfilled are the *title*, the *main-file-key*, and the *summary*. These roles are dealt with by the last three commands in Figure 8.

Automatically Generated Documentation

As a final example of the capabilities of KBEmacs, Figure 12 shows an automatically generated comment for the program UNIT_REPAIR_REPORT. The comment is in the form of an outline. The first line specifies the top level component in the program. The subsequent lines describe how the major roles in this component have been filled in. The comment is constructed based on the components that were used to create the program.

```
-- The procedure UNIT_REPAIR_REPORT is a simple_report:
--   The file_name is "report.txt".
--   The title is ("Report of Repairs on Unit " & UNIT_KEY).
--   The enumerator is a chain_enumeration.
--      It enumerates the chain records in REPAIRS starting from the
--      the header record indexed by UNIT_KEY.
--   The column_headings are a print_repair_headings.
--      It prints headings for printing repair records.
--   The print_item is a print_repair.
--      It prints out the fields of REPAIR.
--   There is no summary.
```

Figure 12: Automatically generated comment for UNIT_REPAIR_REPORT.

The comment generation capability currently supported by KBEmacs is only intended as an illustration of the kind of comment that could be produced. There are many other kinds of comments containing either more or less information that could just as well have been produced. For example, KBEmacs could easily include a description of the inputs and outputs of the program in the comment. The form of comment shown was chosen because it contains a significant amount of high-level information which is not explicit in the program

code. As a result, it should be of genuine assistance to a person who is trying to understand the program.

A key feature of the comment in Figure 12 is that, since it is generated from the knowledge underlying the program, it is guaranteed to be complete and correct. In contrast, much of the program documentation one typically encounters has been rendered obsolete by subsequent program modifications. Although it is not currently supported, it would be easy for KBEmacs to generate a new program comment every time a program was modified.

The fact that KBEmacs can generate the comment shown highlights the fact that KBEmacs always maintains a plan for the program being edited and that this plan contains complete information about what components were used to construct it. Although this is not illustrated above, this gives the approach taken by KBEmacs significant leverage on program maintenance as well as program construction.

Future Directions of the Programmer's Apprentice Project

The work described above is being extended in several directions. To start with, work is progressing rapidly toward the implementation of the topmost layers of CAKE. When CAKE is completed it will be used as the basis for a new demonstration system called the Synthesis Apprentice [24].

The Synthesis Apprentice will incorporate all of the capabilities of KBEmacs. In addition, since it will contain a general-purpose reasoning module (CAKE), it will be able to assist in a greater portion of the programming process. With the assistance of the programmer, the Synthesis Apprentice will be able to create programs based on detailed low-level specifications similar to the comment in Figure 12 As part of this, the Synthesis Apprentice will be able to deduce implicit design decisions which follow from explicit decisions made by the user. The Synthesis Apprentice will also be able to detect many kinds of specification errors.

Work has also begun on a separate system call the Requirements Apprentice[25]. The Requirements Apprentice will assist an analyst in the creation and modification of software requirements. Productivity will be enhanced by allowing an analyst to rapidly build up a requirement by combining standard requirements fragments. Reliability will be enhanced through the use of general-purpose reasoning to detect contradictions and ambiguities in the evolving requirement. Like the Synthesis Apprentice, the Requirements Apprentice will be based on CAKE. However, significant extensions will have to be made in the Plan Calculus so that it can represent reusable requirements components as well as reusable program components.

The long term goal is for the Requirements Apprentice to link up with the Synthesis Apprentice providing support for the entire programming process. However, as currently envisioned, there is a significant gap between the capabilities of these two systems. This gap corresponds to high-level system design. Work on the Synthesis Apprentice will assume that a high-level design has already been obtained and focus on the problem of detailed design.

References

[1] A. Aho, J. Hopcroft, and J. Ullman, *The Design and Analysis of Computer*

Algorithms, Addison-Wesley, 1974.

[2] R. Balzer, "Transformational Implementation: An Example", *IEEE Trans. on Software Eng.*, 7(1), January 1981.

[3] D. Barstow, "An Experiment in Knowledge-Based Automatic Programming", *Artificial Intelligence*, 12:73–119, (reprinted in [23]), 1979.

[4] S. Basu and J. Misra, "Some Classes of Naturally Provable Programs", *2nd Int. Conf. on Software Eng.*, San Francisco CA, October 1976.

[5] M. Broy and P. Pepper, "Program Development as a Formal Activity", *IEEE Trans. on Software Eng.*, 7(1), (reprinted in [23]), January 1980.

[6] T. Cheatham, "Reusability Through Program Transformation", *IEEE Trans. on Software Eng.*, 10(5):589–594, (reprinted in [23]), September 1984.

[7] J. Darlington, "An Experimental Program Transformation and Synthesis System", *Artificial Intelligence*, 16:1–46, (reprinted in [23]), 1981.

[8] J. Dennis, "First Version of a Data Flow Procedure Language", *Proc. of Symposium on Programming*, Institut de Programmation, University of Paris, April 1974.

[9] S. Gerhart, "Knowledge About Programs: A Model and Case Study", in *Proc. of Int. Conf. on Reliable Software*, June 1975.

[10] J. Goguen, J. Thatcher, and E. Wagner, "An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types", *Current Trends in Programming Methodology 4*, R. Yeh *ed.*, Prentice-Hall, 1978.

[11] C. Green, "Theorem Proving by Resolution as a Basis for Question-Answering Systems", *Machine Intelligence 4*, D. Michie and B. Meltzer, Eds., Edinburgh University Press, Edinburgh, Scotland, 1969.

[12] J. Guttag, E. Horowitz, and R. Musser, "Abstract Data Types and Software Validation", *Comm. of the ACM*, 21(12):1048–1064, December 1978.

[13] Y. Ianov, "The Logical Schemes of Algorithms", in *Problems of Cybernetics*, 1, Pergamon Press, New York, 1960.

[14] D. Knuth, *The Art of Computer Programming*, Volumes 1–3, Addison-Wesley, 1968–1973.

[15] B. Liskov, *et. al.*, "Abstraction Mechanisms in CLU", *Comm. of the ACM*, 20(8):564–576, August 1977.

[16] B. Liskov and S. Zilles, "An Introduction to Formal Specifications of Data Abstractions", in *Current Trends in Programming Methodology 1*, R. Yeh *ed.*, Prentice-Hall, 1977.

[17] Z. Manna, *Mathematic Theory of Computation*, McGraw-Hill, 1974.

[18] Z. Manna and R. Waldinger, "A Deductive Approach to Program Synthesis", *ACM trans. on Programming Languages and Systems*, 2(1):90-121, (reprinted in [23]), January 1980.

[19] Z. Manna and R. Waldinger, "Problematic Features of Programming Languages: A Situational-Calculus Approach; Part I: Assignment Statements", Stanford Univ., Weizmann Institute and the Artificial Intelligence Center, August 1980.

[20] D. McAllester, *Reasoning Utility Package User's Manual*, MIT/AIM-667, April 1982.

[21] C. Rich, "A Formal Representation for Plans in the Programmer's Apprentice", *Proc. of 7th Int. Joint Conf. on Artificial Intelligence*, Vancouver, Canada, (reprinted in [23]), August 1981.

[22] C. Rich, "The Layered Architecture of a System for Reasoning about Programs", *Proc. of the 9th Int. Joint Conf. on Artificial Intelligence*, Los Angeles CA, August 1985.

[23] C. Rich and R. Waters *eds.*, *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, Los Altos CA, 1986.

[24] C. Rich and R. Waters, *A Scenario Illustrating a Proposed Program Synthesis Apprentice*, MIT/AIM-933, January 1987.

[25] C. Rich, R. Waters, and H. Reubenstein, "Toward a Requirements Apprentice", *Proc. Fourth Int. Workshop on Software Specification and Design*, Monterey CA, April 1987.

[26] T. Standish, D. Harriman, D. Kibler, and J. Neighbors, *The Irvine Program Transformation Catalogue*, U. of CA at Irvine, 1976.

[27] R. Waters, "The Programmer's Apprentice: A Session With KBEmacs", *IEEE Trans. on Software Engineering*, 11(11):1296–1320, (reprinted in [23]), November 1985.

[28] N. Wirth, *Systematic Programming, An Introduction*, Prentice-Hall, 1973.

[29] W. Wulf, R. London, and M. Shaw, "An Introduction to the Construction and Verification of Alphard Programs", *IEEE Trans. on Software Eng.*, 2(4):253–265, December 1976.

[30] *Military Standard Ada Programming Language*, ANSI/MIL-STD-1815A-1983, U.S. Government Printing Office, February 1983.

[31] *Lisp Machine Documentation*, (release 4), Symbolics, Cambridge MA, 1984.

CS-TR Scanning Project
Document Control Form

Date : 5/26/95

Report # AIM-954

Each of the following should be identified by a checkmark:
Originating Department:

☒ Artificial Intellegence Laboratory (AI)
☐ Laboratory for Computer Science (LCS)

Document Type:

☐ Technical Report (TR) ☒ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 28(34-IMAGES)
Not to include DOD forms, printer instructions, etc... original pages only.

Originals are:
☒ Single-sided or
☐ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☒ Laser Print
☐ InkJet Printer ☐ Unknown ☐ Other:_____

Check each if included with document:

☒ DOD Form (2) ☐ Funding Agent Form ☐ Cover Page
☐ Spine ☐ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages(by page number):_____

Photographs/Tonal Material (by page number):_____

Other (note description/page number):

Description : Page Number:

(A) IMAGE MAP (1) UN#'ED TITLE PAGE
(2-28) PAGES #'ED 1-27
(29-31) SCANCONTROL, DOD(2)
(32-34) TRGT5(3)

(B) CUT & PASTE FIGS ON PAGES 4, 11, 12, 16

Scanning Agent Signoff:

Date Received: 5/26/95 Date Scanned: 6/8/95 Date Returned: 6/8/95

Scanning Agent Signature: Michael W. Cook

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: AIM-954	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER: AD-A185843	
4. TITLE (and Subtitle): Formalizing Reusable Software Components In The Programmer's Apprentice	5. TYPE OF REPORT & PERIOD COVERED: A.I. Memo
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Charles Rich and Richard C. Waters	8. CONTRACT OR GRANT NUMBER(s): NSF# MCS-7912179 ONR# N00014-85-K-0124 (ARPA) IBM (no#) and SPERRY (no#)
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Artificial Intelligence Laboratory 545 Technology Square Cambridge, MA 02139	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, VA 22209	12. REPORT DATE: February 1987 13. NUMBER OF PAGES: 28
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office): Office of Naval Research Information Systems Arlington, VA 22217	15. SECURITY CLASS. (of this report): UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Reuse
Programmer's Apprentice
Software Components
Plan Calculus

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

There has been a long-standing desire in computer science for a way of collecting and using libraries of standard software components. Unfortunately, there has been only limited success in actually doing this. The lack of success stems not from any resistance to the idea, nor from any lack of trying, but rather from the difficulty of choosing an appropriate formalism for representing components.

For a formalism to be maximally useful, it must satisfy five [cont.on reverse]

BLOCK 20, ABSTRACT [continued from front]

key desiderata: expressiveness, convenient combinability, semantic soundness, machine manipulability, and programming language independence. The Plan Calculus formalism developed as part of the Programmer's Apprentice project satisfies each of these desiderata quite well. It does this by combining the ideas from flowchart schemas, data abstraction, logical formalisms, and program transformations.

The efficacy of the Plan Calculus has been demonstrated in part by a porottype program editor called the Knowledge-based Editor in Emacs. This editor makes it possible for a programmer to construct a program rapidly and reliably by combining components represented as plans.

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**